VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Vizualizace izočár s využitím GPU

GPU Based Visualization

2010 David Fiedor

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **David Fiedor**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Vizualizace izočar s využitím GPU
GPU Based Visualization

Zásady pro vypracování:

Moderní hardware, konkrétně grafické procesory (GPU) umožňují řešit specifické úlohy v krátším čase, než běžně dostupné procesory (CPU). Cílem práce je vyzkoušet využití GPU na konkrétní úloze zpracování 2D obrazu a konstrukci izočar.

Hlavní body práce:
1. Specifikace řešení 2D úloh na GPU jejich přínosů oproti řešení na CPU.
2. Návrh a implementace generování izočar s využitím technologie CUDA.
3. Návrh a implementace vizualizace proudění.
4. Výkonostní testy.

Seznam doporučené odborné literatury:

Podle pokynů vedoucího bakalářské práce.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Petr Gajdoš, Ph.D.**

Datum zadání: 20.11.2009
Datum odevzdání: 07.05.2010

doc. Dr.Ing. Eduard Sojka
vedoucí katedry

prof. Ing. Ivo Vondrák, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 5. května 2010 .....

Považuji za svou milou povinnost poděkovat panu Ing. Petru Gajdošovi, Ph.D. za odborné a organizační vedení při zpracování této práce.

Abstrakt

Tato bakalářská práce se zabývá využitím moderní grafické karty s technologií CUDA firmy NVIDIA pro generování izočár. Součástí bakalářské práce je návrh a implementace aplikace, která z daného obrázku (například výškové mapy) vypočítá izočáry a původní obrázek s vygenerovanými izočárami exportuje do zvoleného formátu pro vektorovou grafiku. Technologie CUDA se vyznačuje paralelním zpracováním jednoduchých i složitějších výpočtů pomocí moderní grafické karty, jejíž jádro je tvořeno mnoha procesory. Díky tomu mohou být tyto výpočty zpracovány několikanásobně rychleji než by tomu bylo při zpracování na jakémkoli procesoru současnosti a právě proto tato práce zahrnuje i zpracování výkonnostních testů a jejich porovnání.

Klíčová slova: CUDA, generování izočár, výpočty pomocí GPU, algoritmus marching squares

Abstract

This baccalaureate thesis deals with usage modern graphic card with CUDA technology of firm NVIDIA for generating isolines. Part of the baccalaureate thesis is design and implementation of application, which from given picture (for example height map) will calculate isolines and original picture with generated isolines exports to the selected format for vector graphic. CUDA technology features by multiprocessing of simple and also more complex calculations by force of modern graphic card, whose core is formed by many processors. Thanks that these calculations can be processed many times faster than that would be by processing on any present processor and that's just it this work also includes elaboration performance tests and theirs comparison.

Keywords: CUDA, isolines generating, GPU using calculations, marching squares algorithm

Obsah

1 Úvod **4**

2 Technologie CUDA **5**
2.1 Úvod do výpočtů pomocí GPU 5
2.2 Architektura CUDA 5
2.3 Základní pojmy programové části 8

3 Algoritmus Marching squares **11**
3.1 Marching cubes 11
3.2 Marching squares 11

4 Rozbor zadaného problému **15**
4.1 Načtení obrázku a vytvoření matice z hodnot pixelů 15
4.2 Generování izočár 15
4.3 Export izočár do vektorové grafiky 17

5 Programová část **18**
5.1 Main 18
5.2 IsoLines 19
5.3 StoragePoint 20
5.4 StorageLine 20
5.5 PLStorage 20

6 Výkonnostní testy **23**
6.1 Úvod 23
6.2 Vstupní parametry testů 23
6.3 Testovací sestava 24
6.4 Zhodnocení výsledků testů 24
6.5 Přehled výsledků testů 25

7 Příklady použití aplikace **26**

8 Závěr **28**

9 Reference **30**

Přílohy **30**

A Grafy výsledků testů **31**

Seznam obrázků

1 Srovnání výkonu nVidia GPU a Intel CPU. [6] 5
2 Uspořádání bloků na různých GPU. 6
3 Rozdělení pamětí GPU. [5] 7
4 16 případů rozmístění vnitřních bodů. 12
5 Nejednoznačné případy. 13
6 Náhodná buňka z mřížky. 13
7 Propojení výpočetní matice a matice s hodnotami vrcholů. 16
8 Vstupní výšková mapa 26
9 Výstupní obrázek s velikostí kroku 4 27
10 Výstupní obrázek s velikostí kroku 16 27
11 Srovnání pamětí GPU při různých rozměrech obrázku. 31
12 Srovnání GPU a CPU při různých rozměrech obrázku. 31
13 Srovnání pamětí GPU při různých velikostech kroku. 32
14 Srovnání GPU a CPU při různých velikostech kroku. 32

Seznam výpisů zdrojového kódu

1 Vytvoření a kopírování pole v CUDA aplikaci 9

1 Úvod

Hlavními tématy této bakalářské práce bylo vytvoření funkční aplikace využívající v hlavní výpočetní části technologii CUDA a následné provedení výkonnostních testů, které by porovnávaly dobu běhu aplikace při zpracování výkonnostně náročných výpočtů za pomocí grafické karty a čas vykonávání naprosto stejných výpočtů, avšak pouze s využitím procesoru. Daná aplikace má přijímat zvolený formát obrázku výškové mapy, načíst hodnoty jednotlivých pixelů obrázku, následně tyto hodnoty použít k výpočtu izočár, což jsou čáry na mapě nebo v grafu spojující místa stejných hodnot dané veličiny. Nakonec se vygenerované izočáry vyexportují do zvoleného formátu vektorové grafiky.

První část tohoto dokumentu je věnována popisu teorie potřebné k pochopení dalších částí práce. Jedná se o seznámení s historií vývoje grafických čipů a jejich následné využití nejen ke zpracování grafického výstupu počítače, ale také pro výkonnostně náročné výpočty. Je zde srozumitelně vysvětlen základní princip fungování grafického jádra při paralelních výpočtech a také základní pojmy programové části technologie CUDA. Následující kapitola se pak zaobírá teoretickým vysvětlením funkce algoritmu Marching squares a v jejím závěru je k dipozici i názorný příklad pro pochopení způsobu, jakým jsou generovány křivky použitím této metody.

V další části dokumentu je již podrobně popsán daný problém a poté i postup při implementaci aplikace, přesněji jakým způsobem je získán vstupní obrázek a hodnoty jeho pixelů. Následuje popis generování izočár ze získaných hodnot a uložení vypočtených izočár. Ke konci kapitoly je osvětlen způsob exportu vygenerovaných izočár do vektorové grafiky.

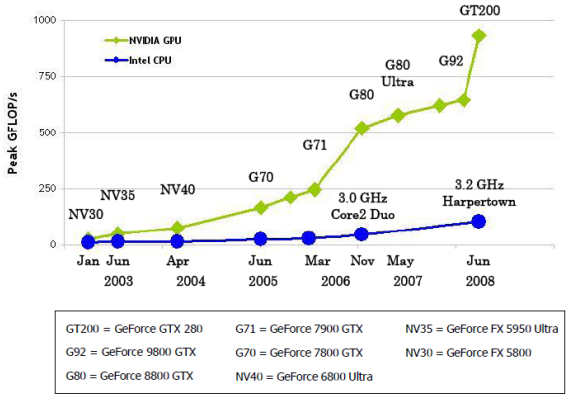
Téměř na konci práce se nachází zpracování výkonnostních testů a podrobné shrnutí jejich výsledků s patřičným komentářem.

V závěru je popsán souhrn výsledků této bakalářské práce a možnosti využití aplikace pro další účely i s nápady na její vylepšení.

2 Technologie CUDA

2.1 Úvod do výpočtů pomocí GPU

Již několik let jsou grafické karty výkonnostně na vyšší úrovni, než jakýkoli moderní procesor posledních let. Přibližně od roku 2003 začaly GPU (Graphic Processing Unit, neboli česky grafický procesor) svým výkonem převyšovat procesory z tohoto období. Jak můžeme vidět na obrázku 1 níže, grafická jádra rok od roku zvyšovala svůj výkonnostní náskok oproti obyčejným procesorům a v současnosti se tento náskok stále více zvyšuje.



Obrázek 1: Srovnání výkonu nVidia GPU a Intel CPU. [6]

Díky tomuto výkonu měly GPU velký potenciál pro náročné výpočty, které by zvládaly spočítat rychleji než běžné procesory. Proto začaly vznikat technologie jako v prvé řadě Compute Unified Device Architecture (dále jen CUDA) od firmy nVidia, Open Computing Language (dále jen OpenCL) vyvíjený konsorciem Khronos, který mimochodem vyvíjí také standard Open Graphics Library (zkratka OpenGL) pro tvorbu počítačové grafiky, nebo ATI Stream společnosti AMD (dříve ATI). I přes to, že OpenCL je mnohem variabilnější jazyk, protože se neomezuje pouze na jádra grafických karet, ale lze ho použít i pro CPU (Central Processing Unit), Cell procesory a jiné platformy, je v současnosti technologie CUDA nejrozšířenější. Tento fakt je dán tím, že CUDA byla vyvinuta ze všech výše zmíněných technologií jako první a společnost nVidia ji implementuje téměř do všech svých nových grafických karet.

2.2 Architektura CUDA

Z výše zmíněného textu tedy vyplývá, že technologie CUDA slouží k provádění výpočtů pomocí grafických karet společnosti nVidia. Abych byl přesný, jedná se o grafické karty s jádrem G80 a vyšším a to jak v rámci série GeForce, tak i sérií Quadro a Tesla. Tato moderní grafická jádra jsou tvořena několika multiprocesory a každý multiprocesor obsahuje 8 procesorů. To znamená, že v této době nejmodernější jádro GT200b s třiceti multiprocesory obsahuje 240 procesorů, které dokážou současně provádět dané výpočty. To zaručuje mnohonásobné zrychlení výpočtů náročných aplikací, které se používají

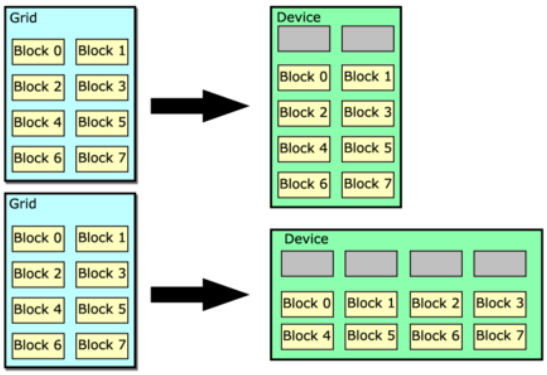
například v odvětví vědy, lékařského výzkumu nebo ke zpracování videa a počítačové grafiky.

Ovšem pokud nemají všechna grafická jádra stejný počet procesorů, bylo nutné zajistit, aby výsledná aplikace nebyla použitelná pouze pro určité typy grafických karet, ale mohla být spustitelná na všech jádrech podporujících technologii CUDA. Proto byly vymyšleny v rámci programového kódu logické celky, které jsou pak dle daného jádra uspořádány a rovnoměrně rozmístěny mezi dostupné multiprocesory.

Tyto logické celky jsou:

- Grid - zde se uchovává počet spuštěných bloků v rámci jednoho kernelu[1]. Tento celek je dvourozměrný a mohou tak být spuštěny dva gridy najednou. Maximální počet spuštěných bloků: 65 535.
- Block - tento celek představuje počet vláken spuštěných v každém bloku. Blok je třírozměrný, což je užitečné například pro výpočty ve vícerozměrných polích. Maximální počet spuštěných vláken v rámci jednoho bloku: 512.

Obrázek 2 v první části znázorňuje, jak se daný Grid uspořádá v grafickém jádře s dvěma multiprocesory, a druhá část ukazuje stejný Grid, ale uspořádaný pro jádro se čtyřmi multiprocesory.

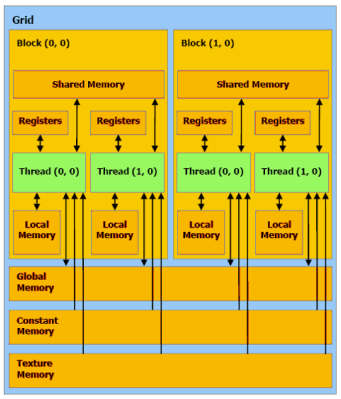


Obrázek 2: Uspořádání bloků na různých GPU.

Zdroj obrázku:
http://www.root.cz/clanky/cuda-optimalizace-pristupu-do-globalni-pameti/

[1]Kernel je metoda spouštěná na GPU

Dále si představíme architekturu z pohledu pamětí grafického jádra. Jak je vidět na obrázku 3, rozdělení pamětí je velice podobné klasickému procesoru. Jsou tu však jisté rozdíly, a proto jednotlivé typy pamětí popíši blíže.



Obrázek 3: Rozdělení pamětí GPU. [5]

Typy pamětí:

1. Registers (registry)
   - nejrychlejší paměť, kterou grafické jádro poskytuje
   - přístup k registrům má pouze dané vlákno
   - umístění - čip jádra
2. Shared memory (sdílená paměť)
   - za určitých okolností může být stejně rychlá jako registry
   - do této paměti mají přístup všechna vlákna, která jsou ve stejném bloku
   - umístění - čip jádra
3. Global memory (globální paměť)
   - může být až 150-krát pomalejší než registry
   - globální paměť je především určena pro přenášení dat mezi grafickou kartou a hostitelským zařízením, je tedy přístupná pro všechna vlákna i bloky grafického jádra
   - umístění - Dynamic Random Access Memory (dále jen DRAM)
4. Local memory (lokální paměť)
   - i když je tato paměť na obrázku zobrazena samostatně, jedná se v podstatě o globální paměť, a proto může také být až 150-krát pomalejší než registry

- používá se například v případech, kdy paměť registrů nedokáže pojmout velké množství proměnných
- přístup k této paměti má pouze dané vlákno
- umístění - DRAM

5. Constant memory (konstantní paměť)
   - je opět součástí globální paměti, ale je také cacheovaná, proto je rychlejší než základní globální paměť
   - slouží pro konstantní hodnoty, které jsou používány všemi vlákny
   - zapisovat hodnoty do této paměti může pouze hostitelské zařízení a číst hodnoty mohou všechna vlákna
   - umístění - DRAM
6. Texture memory (paměť pro textury)
   - součást globální paměti a je opět cacheovaná
   - je přizpůsobená pro často prováděné operace jako mapování nebo deformace 2D textur na 3D polygonální modely
   - stejné pravidla přístupu jako u konstantní paměti
   - umístění - DRAM

2.3 Základní pojmy programové části

Nyní, když alespoň obecně víme, jak vypadá architektura technologie CUDA, vysvětlím základní principy psaní programového kódu spustitelného na grafickém jádře. CUDA Software development kit (dále jen SDK) podporuje relativně velké množství programovacích jazyků a operačních systémů. Z programovacích jazyků se jedná jmenovitě o Fortran, OpenCL, C a DirectCompute. Z operačních systémů zase Windows, Linux a Mac OS X (podpora Mac OS X byla přidána až ve verzi CUDA SDK 2.0). Programovat CUDA aplikace lze i v dalších technologiích jako Java, .NET nebo Python, ale až po doinstalování knihoven takzvaných výrobců třetích stran. V programu k této bakalářské práci jsem použil jazyk C, a proto základní principy programování CUDA aplikací budou také v tomto jazyce.

Kód, který je určen pro vykonávání na grafické kartě, se zapisuje do souboru s příponou *cu*. Metoda vykonávaná na grafickém jádře se v terminologii CUDA nazývá kernel. Aby bylo při překladu zřejmé, že se jedná o kernel a má se tedy kompilovat pomocí kompilátoru CUDA SDK, před název kernelu a jeho návratový typ se zapíše klíčové slovo `__global__`. Samotný kernel se pak spouští uvedením názvu kernelu, následuje speciální syntaxe `<<<pocetBloku, pocetVlaken, velikostSdilenePameti>>>` a za touto syntaxí následují v kulatých závorkách parametry kernelu. První parametr speciální syntaxe udává počet spouštěných bloků, druhý parametr počet vláken v každém spuštěném bloku a poslední parametr je volitelný. Udává velikost sdílené paměti v bytech a používá se pouze

v případě, že chceme použít tuto paměť. Proměnné pro počet bloků a vláken jsou dvou-rozměrné, respektive tří-rozměrné, vektory, ale pokud nám pro spuštění kernelu postačuje pouze jedna složka vektoru, mohou být tyto proměnné celočíselné (tedy typu **int**).

Jelikož se kernel spouští ve více vláknech, která jsou navíc umístěna v několika blocích, musíme vědět, v jakém vlákně nebo bloku se spuštěný kernel nachází nebo kolik vláken či bloků bylo celkem definováno. K tomuto účelu slouží několik proměnných:

- threadIdx - udává index vlákna (číslováno od nuly), ve kterém je aktuálně spuštěn kernel. Nejčastěji využívaná proměnná typu dim3, což je tří složkový vektor. Jak bylo uvedeno v předcházející podkapitole a v textu výše, velikost bloku (tedy počet vláken v bloku) je trojrozměrná hodnota a slouží ke snadnější práci s vícerozměrnými poli. V praxi nemusíme vždy používat všechny složky vektoru, ale můžeme použít například pouze jednu. I v takovém případě však musíme uvést, ze které složky požadujeme identifikační číslo vlákna (threadIdx.x). Obdobně přistupujeme k hodnotám jednotlivých složek i u všech ostatních proměnných.
- blockDim - udává velikost bloku (počet vláken v bloku), proměnná typu dim3.
- blockIdx - udává index bloku, proměnná typu dim2 (dvou-rozměrný vektor).
- gridDim - udává počet bloků v daném gridu, proměnná typu dim2.

Parametry kernelu se zadávají obdobně jako u klasické metody v jazyce C. Pokud však chceme v paměti grafické karty alokovat pole (lineární paměť), musíme k tomuto účelu využít metodu z knihovny CUDA, která je velmi podobná metodě pro alokaci pole v jazyce C. Pokud chceme nakopírovat data uložená v poli v operační paměti do pole, které je alokováno v paměti grafické karty, nebo chceme data nakopírovat opačným směrem, opět je pro tento případ vytvořena metoda v knihovně CUDA SDK. Výpis 1 názorně představuje nejpoužívanější metody knihovny CUDA SDK.

```
...
int *h_pole, *d_pole;
h_pole = (int *)malloc(5 * sizeof(int));
cudaMalloc((void**)&d_pole, 5 * sizeof(int));

naplnPole<<<1, 5>>>(d_pole);
cudaThreadSynchronize();

cudaMemcpy(h_pole, d_pole, 5*sizeof(int), cudaMemcpyDeviceToHost);
cudaFree(d_pole);
...
```

Výpis 1: Vytvoření a kopírování pole v CUDA aplikaci

Poznámka 2.1 Pro přehlednost a lepší orientaci v kódu doporučuji používat prefixy ukazatelů, pro které chceme alokovat paměť (h_ pro ukazatel alokovaný v operační paměti a d_ pro ukazatel alokovaný v paměti grafické karty).

Nejprve si deklarujeme pointery (ukazatele) na jednotlivá pole. Následně alokuji místo v operační paměti pro pole o velikosti 5 a poté alokuji místo pro totožné pole v globální paměti grafické karty. Zavolá se kernel naplnPole, který nám naplní pole v paměti grafické karty nějakými hodnotami. Metoda cudaThreadSynchronize() zajistí, že program nebude pokračovat ve vykonávání následujícího kódu, dokud nebudou všechna vlákna spuštěná pro daný kernel ukončena. Metoda cudaMemcpy požaduje vždy čtyři parametry a to: cílový ukazatel, výchozí ukazatel na pole, velikost přenášených dat a směr přenášení dat (pro kopírování dat z operační paměti do globální paměti grafické karty se používá příznak cudaMemcpyHostToDevice). Po překopírování dat je místo alokované v paměti grafické karty uvolněno.

Existují také metody pro alokaci a kopírování dat určené pro dvourozměrná a trojrozměrná pole. Jsou to metody cudaMallocPitch, cudaMalloc3D pro alokaci a cudaMemcpy2D, cudaMemcpy3D pro kopírování dat.

Pro lepší pochopení paralelního programování doporučuji nastudovat alespoň jednu z publikací [2], [3] nebo [1].

3 Algoritmus Marching squares

3.1 Marching cubes

Algoritmus Marching cubes (v překladu "Pochodující krychle") je metoda používaná k povrchovému vyjádření dat zaznamenaných ve voxelové podobě. Jedná se tedy o popis daného voxelového objektu pomocí plochy navzájem propojených n-úhelníků. Nejčastěji je využívána pro vyjádření medicínských dat, tedy ve spojení s dobrým prahováním objektu, pro snadnější odlišení jeho částí.[4]

Poznámka 3.1 Voxel - jedná se o složeninu anglických slov volumetric, česky "objemový", a pixel. Je to částice objemu, představující hodnotu v pravidelné mřížce tří dimenzionálního (dále jen 3D) prostoru počítačové grafiky. Je to v podstatě analogie k pixelu, který reprezentuje dvou dimenzionální (dále jen 2D) grafiku.

Jelikož ve své práci tento algoritmus nevyužívám a uvádím ho zde jen proto, že algoritmus Marching squares vychází z tohoto algoritmu, nebudu již metodu Marching cubes blíže popisovat.

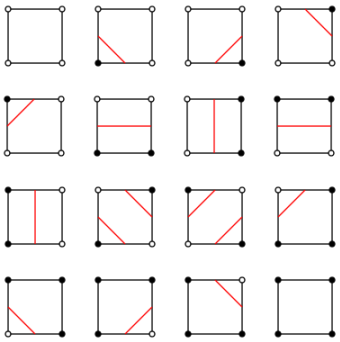
3.2 Marching squares

3.2.1 Princip

Princip algoritmu Marching squares (česky "Pochodující čtverce") je velice podobný principu výše zmíněného algoritmu Marching cubes. Hlavní rozdíl mezi těmito algoritmy je v tom, že Marching cubes se používá v trojrozměrném prostoru a Marching squares zase v dvojrozměrném prostoru. Jak podrobněji uvedu níže, obě metody mají několik případů, podle kterých se pak rozhoduje, jak se má algoritmus dále chovat. V rámci Marching squares se jedná přesně o šestnáct případů, kdežto Marching cubes jich má díky 3D prostoru daleko více a to 256.

Mějme tedy mřížku s indexy i a j, které odpovídají krokům na ose x a y, o velikosti m a n, kde index i nabývá hodnot od nuly až po $m - 1$ a index j hodnot od nuly až po $n - 1$. Tyto indexy nám označují jednotlivé vrcholy dané mřížky. Každý vrchol této mřížky je ohodnocen určitou hodnotou. Dále si nadefinujeme hodnotu h, která nám udává reálnou velikost posuvu při posunu v indexu o 1 a platí jak pro index i, tak pro index j. Nakonec máme danou určitou referenční hodnotu, se kterou jednotlivé vrcholy mřížky porovnáváme. Tuto hodnotu si nazveme například f_0 a hodnotu vrcholů mřížky jako f_{ij}. Pomocí referenční hodnoty f_0 hledáme v naší mřížce křivku, která této hodnotě odpovídá. Následně procházíme mřížku po jednotlivých buňkách (procházíme pouze vrcholy mřížky, ale vždy porovnáváme s referenční hodnotou čtyři vrcholy, které nám tvoří buňku mřížky) a rozhodujeme, zda vrcholy dané buňky vyhovují referenční hodnotě nebo ne. V případě, že porovnávaný vrchol splňuje podmínku $f_{ij} \leq f_0$, pak tento vrchol označujeme v rámci dané buňky jako vnitřní vrchol. Jestliže porovnávaný vrchol tuto podmínku nesplňuje, pak jej označujeme jako vnější vrchol. Pokud je jeden vrchol označen jako vnitřní a druhý jako vnější a zároveň tyto dva vrcholy leží na stejné hraně buňky,

vyplývá nám z této situace, že jeden z bodů hledané křivky leží někde mezi těmito vrcholy. Takto si označíme všechny čtyři vrcholy porovnávané buňky a dostaneme index, který nám udává, o který z šestnácti možných případů ohodnocení bodů se jedná. Obrázek 4 níže zobrazuje všech šestnáct případů rozmístění vnitřních bodů. Vrcholy s bílou výplní udávají vnější body a vrcholy s černou výplní naopak vnitřní body. Červená čára označuje, kudy vede hledaná křivka.



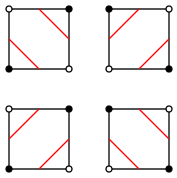
Obrázek 4: 16 případů rozmístění vnitřních bodů.

Nejvýhodnější metoda pro získání indexu celé buňky je ohodnocení vrcholů pomocí dvojkové soustavy. Pokud si vrcholy buňky označíme jako A, B, C a D (levý dolní, pravý dolní, pravý horní a levý horní vrchol), pak index získáme jako
$index = a \cdot 2^0 + b \cdot 2^1 + c \cdot 2^2 + d \cdot 2^3$, kde a, b, c, d jsou hodnoty 0 nebo 1 podle toho, zda body A, B, C, D jsou vnitřními body nebo ne. Tímto způsobem dostaneme index v rozmezí hodnot $<0, 15>$. Podle tohoto získaného indexu pak z takzvané tabulky úseček (nebo jinak definovaných bodů hledané křivky pro daný případ) zjistíme, na kterých hranách, respektive mezi kterými vrcholy porovnávané buňky, se nachází koncové body hledané křivky. Když získáme polohu jednotlivých bodů křivky, stačí již jen vypočíst vzdálenost těchto bodů od vrcholů buňky a to jako $h/2$. Takto vypočtená vzdálenost však není velmi přesná, a proto se spíše používá lineární interpolace.

3.2.2 Nejednoznačnosti algoritmu

Při vypočítávání křivek pomocí algoritmu Marching squares (stejně jako u Marching cubes) mohou nastat za určitých podmínek případy, kterým říkáme nejednoznačné nebo dvojsmyslné (anglicky nazývané jako ambiguous cases). Na obrázku 5 jsou tyto případy demonstrovány. V praxi nelze jednoznačně určit, zda použijeme k vykreslení křivek případy z prvního či z druhého řádku. Proto se musíme již při implementaci algoritmu rozhodnout, který z těchto řádků použijeme. Pokud nebudeme oba řádky kombinovat, můžeme použít kterýkoli řádek a výsledek bude obdobný. Jestliže zvolíme případy z druhého řádku, vymezené oblasti, které vyhovují hodnotě f_0, budou souvislé. Když zvolíme pro implementaci první řádek, vymezené oblasti vyhovující hodnotě f_0

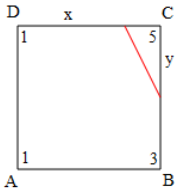
budou odtržené a naopak zbylá oblast (nevyhovující referenční hodnotě) bude souvislá. Pro svou aplikaci jsem zvolil právě případy z druhého řádku.



Obrázek 5: Nejednoznačné případy.

3.2.3 Názorný příklad

Pro snadnější pochopení principu algoritmu Marching squares jsem si připravil následující příklad. Obrázek 6 ukazuje náhodnou buňku z mřížky s již vygenerovanou výslednou křivkou s využitím lineární interpolace. Následujícím postupem si ukážeme, jak zjistit velikosti úseček x a y, tedy vzdálenosti koncových bodů hledané křivky od výchozích vrcholů dané buňky.



Obrázek 6: Náhodná buňka z mřížky.

Mějme hodnotu $f_0 = 4$. Nejprve získáme hodnoty vrcholů pro porovnávání s f_0:

- $A = f_{ij+1} = 1$
- $B = f_{i+1j+1} = 3$
- $C = f_{i+1j} = 5$
- $D = f_{ij} = 1$

Z těchto vrcholů za pomocí porovnávací funkce $A(B, C, D) \leq f_0$ se dozvíme, že pouze jediný vrchol zkoumané buňky je označen jako vnější a to vrchol C. Ohodnotíme všechny vnitřní vrcholy dle výše uvedeného schématu dvojkové soustavy a dostaneme hodnotu $index = 11$. Podle něj vyhledáme v tabulce úseček, kde se nacházejí koncové body křivky.

Takto zjistíme, že hledané body křivky se nachází mezi vrcholy DC a CB. Následně lineární interpolací vypočteme vzdálenost mezi těmito body:

$$x = \frac{f_0 - D}{C - D} \cdot h = \frac{4 - 1}{5 - 1} \cdot h = \frac{3h}{4}$$

$$y = \frac{f_0 - C}{B - C} \cdot h = \frac{4 - 5}{3 - 5} \cdot h = \frac{h}{2}$$

Tento postup opakujeme pro všechny ostatní buňky v mřížce, čímž vznikne jedna nebo více souvislých oblastí, které vyhovují referenční hodnotě f_0. Všechny sousední buňky, které mají na své společné hraně koncový bod křivky, mají také vždy stejné hodnoty vrcholů na společné hraně. Z toho vyplývá, že i lineární interpolace těchto vrcholů si budou rovny a výsledná křivka bude souvislá. V ideálním případě křivka nebo křivky tvoří uzavřený obrazec. U buněk ležících na okraji mřížky však dochází k případům, kdy díky konci mřížky nemůžeme určit jak křivka pokračuje a výsledná křivka je v těchto místech otevřená.

4 Rozbor zadaného problému

Mým úkolem je na základě výše uvedené teorie vytvořit aplikaci, která pro zjednodušení a přehlednost v tomto dokumentu bude rozdělena do následujících částí:

1. Načtení obrázku a vytvoření matice z hodnot pixelů
2. Generování izočár
3. Export izočár do vektorové grafiky

4.1 Načtení obrázku a vytvoření matice z hodnot pixelů

Jako vstupní soubor jsem si pro mou aplikaci vybral bitmapu (formát obrázku pro ukládání rastrové grafiky) ve stupních šedi, jejíž bitová hloubka činí 8 nebo 24 bitů. Pro základní operace typu načtení obrázku, zjištění šířky a výšky obrázku a jiné, využívám ve svém projektu open source (česky "otevřený zdrojový kód") knihovnu FreeImage (verze 3.13.1), která je šířena pod dvojí licencí GNU General Public License a FreeImage Public License, které umožňují volné užití této knihovny pro open source, respektive komerční, projekty. Více informací zjistíte na adrese <http://freeimage.sourceforge.net/>. Pomocí metody, která využívá této knihovny, tedy načtu danou bitmapu. Tato metoda mi zároveň nastaví proměnné $width$ = šířka bitmapy a $height$ = výška bitmapy.

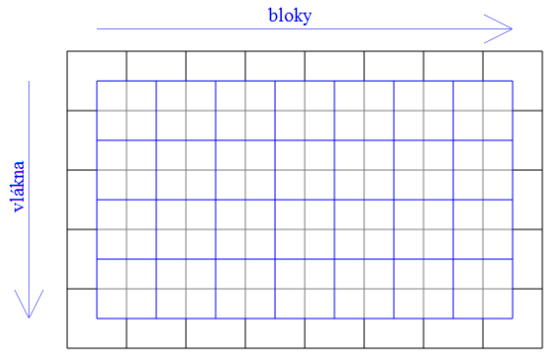
Následně potřebuji získat matici o velikosti $width$ x $height$, do které uložím hodnoty získané z jednotlivých pixelů bitmapy. Jelikož tuto matici bych dále poslal do níže popsaného kernelu pro generování izočár, rozhodl jsem se pro uložení hodnot z bitmapy do matice použít také kernel, ve kterém provedu uložení dat do matice a ukazatel na tuto matici uloženou v paměti grafické karty poté přepošlu do kernelu pro generování izočár.

Poznámka 4.1 I když používám výraz matice, který představuje dvojrozměrné pole, v programu ukládám data pouze do jednorozměrných polí. Výraz matice užívám jen proto, že si lze pak lépe představit popisované pojmy.

4.2 Generování izočár

Pro tuto část aplikace je vytvořen cyklus, který představuje zvyšování testovací (referenční) úrovně o velikost kroku. Testovací úroveň může nabývat hodnot z intervalu <0, 255> (256 stupňů šedí) a krok je nastaven při spuštění programu. V prvé řadě je spuštěn kernel pro generování izočár. Jedním z parametrů tohoto kernelu je i již výše zmíněný ukazatel na pole s načtenými hodnotami jednotlivých pixelů. Tento postup je výhodný, jelikož data zůstávají stále v paměti grafické karty a nemusejí být stále překopírovávány z grafické paměti do operační a naopak. Kernel implementuje algoritmus Marching squares. Pro výpočty používám druhou takzvanou výpočetní matici o velikosti $width - 1 x height - 1$. Tato matice, obrazně řečeno, leží na matici s hodnotami pixelů, která nám tak udává hodnoty vrcholů buněk výpočetní matice. Výpočetní matice v podstatě představuje testovanou mřížku algoritmu Marching squares. Jak toto napojení obou

matic přesně funguje nám demonstruje obrázek 7. Černou, respektive šedou, barvou je znázorněna matice s hodnotami jednotlivých vrcholů výpočetní matice a modrá barva znázorňuje výpočetní matici. Znázorněné šipky představují směr "růstu" spuštěných bloků a vláken. Tuto metodu organizace vláken jsem zvolil proto, že maximální počet vláken spuštěných v rámci jednoho bloku je 512, počet bloků v gridu může být až 65000 a obrázky se většinou vytvářejí naležato nebo jsou čtvercového formátu. Z tohoto důvodu používám vlákna pro procházení sloupců a ne řádků (počet spuštěných bloků je roven šířce výpočetní matice). Pokud je výška výpočetní matice větší než 512, kdy počet vláken již na tuto velikost nestačí, vlákna pak daný výpočet izočár opakují i pro buňky s indexem větším než 511, dokud nespočítají všechny buňky ve svém sloupci.



Obrázek 7: Propojení výpočetní matice a matice s hodnotami vrcholů.

V implementovaném algoritmu Marching squares nepoužívám klasickou lineární interpolaci, jelikož jsou jisté případy, kdy pomocí lineární interpolace nebude vykreslena křivka, ale pouze bod, který však v konečném důsledku nebude vidět. Mou domněnku vysvětlím na následujícím příkladu.

Mějme například buňku s vrcholy A, B, C a D, kde jejich vrcholy mají hodnoty (5, 5, 5, 1), a hodnotu $f_0 = 1$. Když postup získávání křivky zkrátím, po určitých krocích budu vědět, že koncové body hledané křivky se nacházejí mezi vrcholy DA a DC. Po aplikaci lineární interpolace na jednotlivé hodnoty těchto vrcholů získáme hodnoty $x = y = 0$. V takovém případě, jak jsem již zmínil, dostanu pouze bod a ne křivku. Daná buňka pak vypadá, jako by všechny její vrcholy byly vnitřními nebo vnějšími vrcholy. Proto používám následující postup:

1. porovnám hodnoty výchozího a koncového vrcholu
2. vypočítám velikost kroku $add = \frac{h}{vetsiHodnotaVrcholu - mensiHodnotaVrcholu + 1}$
3. zjistím násobek velikosti kroku $multi = vychoziVrchol - f_0$
4. pokud je $multi \leq 0$ provedu $multi = multi \cdot (-1)$ a poté $multi = multi + 1$
5. nakonec vypočítám vzdálenost bodu křivky od výchozího vrcholu $add = add \cdot multi$

Tato metoda sice správně vykresluje krajní křivky, ale není příliš přesná, protože pokud bychom měli stejnou buňku jako jsme použili výše a referenční hodnota by byla $f_0 = 3$, pak by vzdálenost od výchozího bodu měla být $h/2$. V případě mé metody však dochází k vychýlení směrem k vrcholu s vyšší hodnotou. Jelikož je ale v mé aplikaci hodnota $h = 1$ (reprezentuje velikost pixelu), je tato odchylka zanedbatelná.

Po skončení výpočtů kernelu se vygenerované izočáry uloží do úložiště v paměti, kde jsou ukládány i se svými příslušnými body. Jakmile jsou porovnány všechny úrovně, respektive testovací úroveň převýší hodnotu 255, tato část aplikace končí.

4.3 Export izočár do vektorové grafiky

Pro provedení exportu vygenerovaných izočár využívám knihovny LibBoard (verze 0.8.6b), která je šířena pod licencí GNU Lesser General Public License. Bližší informace na adrese <http://libboard.sourceforge.net/>. Tato knihovna umožňuje ze zadaných objektů (čáry, obdélníky, body, a jiné) provádět exporty do formátů PostScript, Fig a SVG (Scalable Vector Graphics).

Export izočár probíhá v těle samostatné metody, která prochází jednotlivé uložené čáry v paměti. Pro každou čáru získá počáteční a koncový bod a jejich souřadnice načte do objektu, který představuje knihovnu LibBoard, v podobě čáry. Když metoda načte všechny izočáry, zavolá se metoda knihovny LibBoard, která všechny načtené čáry uloží do SVG souboru. Protože knihovna LibBoard prozatím nepodporuje vkládání obrázků do SVG souboru, upravil jsem ji tak, aby bylo možné vstupní obrázek importovat do výstupního souboru s vykreslenými izočárami. Je tedy nutné mít originální bitmapu umístěnou ve stejné složce s výstupní SVG souborem, aby se bitmapa zobrazovala jako pozadí.

5 Programová část

Aplikace je napsána v jazyce C++ s využitím knihovny CUDA SDK 2.3, která je určena pro tvorbu kódu spustitelného na grafických kartách s čipem společnosti nVidia. V této části následovně podrobněji popíši jednotlivé části aplikace, jejich metody (kernely) a parametry daných funkcí.

5.1 Main

Hlavní část aplikace, která tvoří jakousi kostru programu, obsahuje pouze kód spustitelný na procesoru a metody pro úvodní inicializaci nebo závěrečný export izočár do SVG formátu.

ParseCommands

- Kontroluje zadané parametry při spouštění programu a u nepovinných parametrů nastavuje jejich výchozí hodnoty.
- Parametry:
  - *argc* - počet zadaných argumentů
  - *argv[]* - pole se zadanými argumenty
  - *path* - název (cesta) souboru s bitmapu výškové mapy
  - *stepLevel* - velikost kroku pro zvyšování testovaných úrovní
  - *textOutput* - pokud je hodnota nastavena na true, ukládá souřadnice vygenerovaných izočár do textového souboru "output.txt"
- návratová hodnota - true v případě, že byly dané argumenty zadány správně

PrintHelp

- Vytiskne nápovědu s výpisem všech podporovaných parametrů pro spuštění aplikace. Metoda je zavolána pokud metoda ParseCommands vrátí hodnotu false.

LoadImage

- Ověřuje zda zadaný soubor je či není bitmapa.
- Parametry:
  - *fileName* - název souboru s bitmapou
  - *width* - šířka bitmapy
  - *height* - výška bitmapy
  - *pitch* - šířka bitmapy bytech (Pro výkonnostní účely je v knihovně FreeImage každý řádek s RGB hodnotami jednotlivých pixelů zarovnán na násobky 16-ti, proto je nutné pro správné procházení hodnot pixelů znát tuto hodnotu. Nelze se orientovat dle hodnoty width!)

  - *bitDepth* - bitová hloubka bitmapy

- návratová hodnota - ukazatel na objekt načtené bitmapy

ExportToSVG

- Prochází uložené izočáry a exportuje je pomocí knihovny LibBoard do SVG souboru vektorové grafiky.
- Parametry:
  - *lines* - objekt s uloženými izočárami a jejich body
  - *width* - šířka bitmapy
  - *height* - výška bitmapy
  - *fileName* - název výstupního SVG souboru
  - *original* - název vstupní bitmapy

5.2 IsoLines

Zde jsou uloženy kernely a jejich inicializační metody.

CheckCUDAError

- Kontroluje, zda při operacích na GPU nedošlo k chybě. Pokud ano, danou chybu vypíše a ukončí aplikaci.
- Parametry:
  - *msg* - označení prováděné operace (například memcpy nebo kernel invocation)

GetPixelValue

- Inicializuje potřebné proměnné pro kernel *GetPixelValueKernel* a poté ho spustí.
  - *h_bits* - pole s RGB hodnotami pixelů
  - *d_imageGrid* - pole pro uložení hodnot z pixelů
  - *width* - šířka bitmapy
  - *height* - výška bitmapy
  - *pitch* - šířka bitmapy v bytech
  - *bitDepth* - bitová hloubka
  - *gridMemSize* - velikost pole *d_imageGrid*
- návratová hodnota - pokud hodnota bitové hloubky není hodnota 8 nebo 24, vrací false, jinak true

IsoLines

- Inicializuje potřebné proměnné pro kernel *IsoLinesKernel* a poté ho spustí.
- Parametry:
  - *d_imageGrid* - pole s uloženými hodnotami pixelů
  - *width* - šířka bitmapy
  - *height* - výška bitmapy
  - *lines* - ukazatel na objekt pro uložení izočár a jejich bodů
  - *levelStep* - velikost kroku testovacích úrovní
  - *textOutput* - pokud je nastavena hodnota true, souřadnice bodů izočár budou uloženy do textového souboru "output.txt"

5.3 StoragePoint

V této třídě jsou uchovávány souřadnice bodu. Dva tyto body pak tvoří čáru. Třída obsahuje takzvané get a set metody, což jsou metody pro nastavení a získání jednotlivých bodů.

- privátní proměnné:
  - *x* - souřadnice x typu float
  - *y* - souřadnice y typu float

5.4 StorageLine

Třída představuje čáru, která uchovává odkazy na body dané čáry formou indexu jednorozměrného vektoru, který ukládá všechny body. Jsou zde také get a set metody pro jednotlivé indexy bodů.

- privátní proměnné:
  - *point1* - index na počáteční bod typu unsigned int
  - *point2* - index na koncový bod typu unsigned int

5.5 PLStorage

Třída, která uchovává v samostatných jednorozměrných vektorech body a čáry. Navíc obsahuje hash tabulku implementovanou jako mapu. Hash tabulka slouží pro rychlejší porovnávání souřadnic vkládaných bodů. Protože vygenerované izočáry ohraničující souvislé oblasti na sebe navazují, každé dvě čáry mají tedy jeden společný bod. Proto je zbytečné ukládat tyto společné body vícekrát.

V případě, že vkládáme nový bod, vytvoří se pomocí hash funkce z jeho souřadnic hash hodnota. Tato hodnota se následně porovnává s hodnotami v hashovací tabulce.

Pokud daný hash v tabulce existuje, znamená to, že daný bod také existuje a není nutné jej ukládat. Pokud ale hodnota hashe v hashovací tabulce neexistuje, uloží se hash do hashovací tabulky a bod do vektoru s uchovávanými body.

Třída však není primárně určená pro ukládání samotných bodů, i když se takto dá použít. Její hlavní funkcí v rámci této aplikace je ukládaní čar. Čára je tvořena dvěma body a výše zmíněný postup ukládání bodu je použit pro oba body vkládané čáry.

- definované typy:
  - *HashTable* - vlastní definovaný typ pro hash tabulku, který je vytvořen z původního typu map<__int64, unsigned int>
- privátní proměnné:
  - *points* - jednorozměrný vektor pro uchovávání bodů typu vector<SotragePoint>
  - *lines* - jednorozměrný vektor pro uchovávání čar typu vector<SotrageLine>

HashPoint

- Vytváří z daných souřadnic hash hodnotu.
- Parametry:
  - *x* - souřadnice x
  - *y* - souřadnice y
- návratová hodnota - hash vytvořený z daných souřadnic

PointExists

- Prochází hashovací tabulku a zjišťuje, zda hash porovnávaného bodu existuje.
  - *hash* - hash hodnota
- návratová hodnota - pokud je nalezen daný hash v tabulce, vrací jeho index, který je totožný s indexem bodu uloženým ve vektoru s uloženými body, jinak vrací hodnotu -1

AddPoint

- Pokud bod s danými souřadnicemi neexistuje, pak tento bod uloží.
- Parametry:
  - *x* - souřadnice x
  - *y* - souřadnice y
- návratová hodnota - index pozice bodu ve vektoru points

AddLine

- Počáteční i koncový bod čáry je uložen a indexy obou bodů pak tvoří čáru, která je uložena do vektoru lines.
- Parametry:
  - *x1* - souřadnice x počátečního bodu
  - *y1* - souřadnice y počátečního bodu
  - *x2* - souřadnice x koncového bodu
  - *y2* - souřadnice y koncového bodu

V třídě se také nachází get metody pro načtení čáry nebo bodu na zadané pozici (pokud je pozice mimo hranice vektoru lines, respektive points, je vrácena hodnota NULL) a načtení velikosti vektoru points nebo lines.

6 Výkonnostní testy

6.1 Úvod

Součástí této bakalářské práce bylo vytvoření výkonnostních testů, které by porovnávaly přínosy zrychlení při provádění výpočetně náročných částí aplikace. Jelikož jsem program vytvářel v několika iteracích, měl jsem možnost porovnat nejen zrychlení provádění kódu na grafické kartě s využitím sdílené paměti oproti zpracování na běžném procesoru, ale mohl jsem do výkonnostních testů zahrnout i srovnání výkonu aplikace při spuštění na grafické kartě pouze s využitím globální paměti.

Svou finální aplikaci jsem tedy pro potřeby testování rozšířil o dříve napsaný kernel, který nevyužíval sdílenou paměť a jeho vstupní parametry jsem upravil tak, aby se daly měnit podmínky testování bez nutnosti opětovné kompilace programu. Jedná se jmenovitě o tyto parametry:

- **/iteration** - za parametrem následuje číslo od 1 do 10, které určuje počet opakování dané konfigurace.
- **/noShareMem** - pokud je nastaven tento parametr, bude spuštěna verze kernelu bez sdílené paměti.

Následně jsem odstranil exportování vygenerovaných izočár do formátu vektorové grafiky SVG a to proto, že tato část programu je vykonávána pouze procesorem,
takže nemá příliš velkou vypovídající hodnotu při porovnávání rychlosti zpracování generování izočár na grafické kartě a procesoru. Navíc export izočár je časově nejnáročnější část aplikace, čili byl odstraněn i z časových důvodů. I přes fakt, že ukládání jednotlivých izočár a jejich bodů do připraveného úložiště je také vykonáváno jen na procesoru, nejedná se v porovnání s exportem o časově náročnou operaci, a proto jsem ji v programu ponechal. Technologie CUDA dokáže emulovat grafickou kartu podporující tuto technologii, kdy veškerý kód aplikace probíhá pouze na procesoru. Tato vlastnost je primárně určena pro případ, kdy vývojář CUDA aplikací nemá přístup ke grafické kartě, která dokáže zpracovat takový program. Já jsem však toho využil pro otestování výkonnosti aplikace při zpracování procesorem. Nemusel jsem tak zbytečně a zdlouhavě přepisovat kód určený pro zpracování na grafické kartě, který by byl použit jen pro účely porovnání výkonnosti.

6.2 Vstupní parametry testů

Jako vstupní testovací obrázek jsem použil obrázek výškové mapy z webové stránky <http://www.letka13.sk/forum/viewtopic.php?f=19&t=1089>.
Protože obrázek byl původně ve formátu JPEG (Joint Photographic Experts Group) a má aplikace vyžaduje bitmapu, převedl jsem tedy tento obrázek pomocí obyčejného Malování, které je součástí operačního systému Windows, do několika bitmapových obrázků s bitovou hloubkou 24 bitů. Tyto testovací obrázky měly velikost 64x64, 128x128, 256x256 a 512x512 pixelů.

Dále jsem vytvořil dvě skupiny testů. První sada testů porovnávala vliv rozměrů obrázku na délku zpracování výpočtů, přičemž velikost kroku zvyšování testovacích úrovní byla nastavena na hodnotu 8. Druhá sada testů zkoumala vztah mezi výpočetním časem aplikace a velikostí kroku testovací úrovně. K testování byl použit obrázek o velikosti 512x512 pixelů. Aby byla získána průměrná hodnota doby běhu programu, nastavil jsem počet opakování pro všechny testy na tři, mimo testu aplikace spuštěné pouze na CPU. Jelikož z prvních zkušeností s testováním jsem vypozoroval, že zpracování celého programu pouze na procesoru trvá dlouhou dobu, u testů výkonu procesoru jsem použil pouze jedno opakování. Nejedená se tedy o průměrnou hodnotu.

6.3 Testovací sestava

Jako testovací sestavu pro výkonnostní testy jsem použil vlastní laptop s následujícími parametry:

- **operační systém** - Microsoft Windows 7
- **procesor** - AMD Turion Ultra ZM-80 (2,1 GHz)
- **operační paměť** - 4GB DDR2
- **grafická karta** - NVIDIA GeForce GT 130M
  - velikost paměti: 512MB
  - počet multiprocesorů: 4
  - počet jader: 32 ($4 \cdot 8$)

6.4 Zhodnocení výsledků testů

Protože rozdíly mezi zpracováním výpočtů na GPU a CPU byly velké a naopak rozdíly mezi GPU s využitím sdílené paměti a GPU s použitím globální paměti byly v řádech milisekund, výsledné grafy jsem rozdělil u obou skupin testů na srovnání výkonu mezi GPU se sdílenou pamětí a CPU a na srovnání výkonu výpočtů pouze na grafických kartách s využitím různých pamětí. Zaměřím se nejprve na zhodnocení testování výkonnosti grafického jádra s použitím sdílené paměti a bez ní. Zde byly pro mne výsledky mírným zklamáním. Očekával jsem od použití sdílené paměti daleko větší zrychlení než jaké jsem naměřil. Rozdíly mezi jednotlivými výpočetními časy byly i v testech s největšími odstupy v řádech milisekund. Přesně 0,067 sekundy u minimální velikosti kroku v případě testů vlivu velikosti kroku a 0,05 sekundy u velikosti obrázku 512x512 v rámci porovnávání vlivu velikosti obrázku. Domnívám se, že větší zrychlení by se projevilo až při zpracování většího množství dat, tedy pokud by byl obrázek větších rozměru než 512x512 pixelů (například 1024x768 a vyšší) a zároveň by byla velikost kroku nastavena na minimální hodnotu.

U testů srovnání výkonnosti GPU a CPU byla situace opačná. Zde mne naopak rozdíl časů zpracování překvapil. Největší rozdíl pro test vlivu velikosti kroku byl naměřen

opět u hodnoty velikosti kroku 4 a to 439,323 sekund, což je 98-mi násobné zrychlení výpočtu GPU oproti CPU. U testu vlivu velikosti obrázku to byl rozdíl 223,010 sekund a 103 krát rychlejší zpracování výpočtu na GPU pro velikost obrázku 512x512. Navíc jsou tyto výsledky dosaženy na grafické kartě pouze s 32-mi jádry, když vezmeme v úvahu, že nejnovější karty mají až 240 jader, rozdíly mohou být daleko větší.

6.5 Přehled výsledků testů

Kvůli přehlednosti a velikosti grafů jsem jejich obrázky umístil do přílohy A. Obrázky 11 a 12 ukazují výsledky k testu vlivu rozměrů obrázků na výpočetní čas aplikace. Obrázky 13 a 14 pak zobrazují naměřené hodnoty pro test vztahu mezi velikostí kroku testovacích úrovní na délku výpočtu programu. První obrázek u obou sad testů vždy znázorňuje výsledky pro srovnání GPU s využitím sdílené a globální paměti, druhý obrázek pak výsledky testů pro srovnání délky výpočtů na grafické kartě a procesoru.

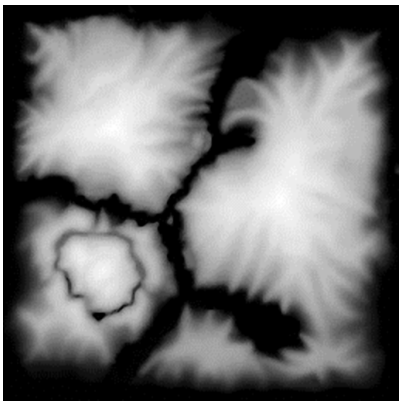
7 Příklady použití aplikace

Program ke svému chodu potřebuje pouze CUDA a FreeImage dynamické knihovny. Autor knihovny pro generování vektorové grafiky LibBoard poskytuje pouze zdrojové soubory a tudíž musí být tyto zdrojové kódy zahrnuty přímo v projektu a kompilují se spolu s ostatními zdrojovými soubory. Aplikace je spouštěna přes příkazovou řádku a vyžaduje pouze jeden povinný parametr a to název bitmapového souboru s výškovou mapou. Součástí zdrojových souborů aplikace jsou také požadované dynamické knihovny i testovací bitmapy. Proto lze vyzkoušet funkce programu, aniž by bylo nutné hledat vhodný obrázek. Program podporuje následující parametry:

- **název souboru** - název bitmapového souboru musí být vždy uveden jako první parametr
- **/levelStep** - nepovinný parametr, udává velikost kroku při zvyšování testovacích úrovní (hodnota z intervalu <1, 256> následuje za tímto parametrem)
- **/textOut** - nepovinný parametr, pokud je nastaven, jednotlivé body křivek budou uloženy do textového souboru s názvem output.txt

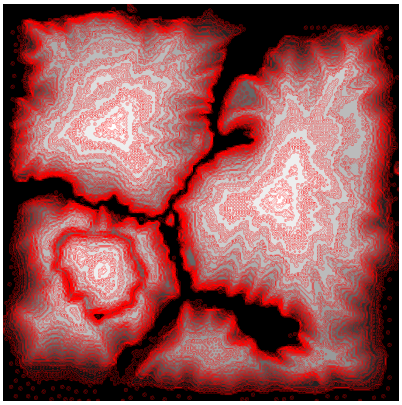
Ve zbylé části této kapitoly přikládám dvě ukázky použití mé aplikace i s náhledy na vstupní a výstupní obrázky. Výstupní obrázky pak demonstrují rozdíly ve vygenerovaných souborech při použití různých hodnot kroků pro zvyšování testovacích úrovní.

Na obrázku 8 je vidět bitmapa s názvem *512x512x24.bmp*, která byla použita pro výkonnostní testy a kterou použiji jako vstupní obrázek i v následujících příkladech použití aplikace.



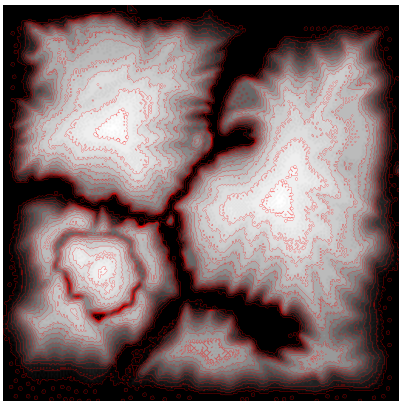
Obrázek 8: Vstupní výšková mapa

Pokud spustíme aplikaci pouze s parametrem, který označuje název vstupní bitmapy, použije se výchozí krok pro zvyšování testovacích úrovní. Ten je implicitně nastaven na hodnotu 4. Na obrázku 9 je pak vidět výstupní obrázek s vypočítanými izočárami. Spuštění z příkazové řádky: *isolines.exe 512x512x24.bmp*



Obrázek 9: Výstupní obrázek s velikostí kroku 4

V případě, že použijeme při spuštění parametr pro nastavení velikosti kroku testovacích úrovní, za ním hodnotu 16 a zároveň použijeme stejnou vstupní bitmapu jako v předchozím příkladu, výstupní SVG soubor bude vypadat stejně jako na obrázku 10. Spuštění z příkazové řádky: *isolines.exe 512x512x24.bmp /stepLevel 16*



Obrázek 10: Výstupní obrázek s velikostí kroku 16

Jak je patrné z obrázků 9 a 10, mezi velikostí kroku a počtem izočár platí nepřímá úměra: čím vyšší je hodnota kroku, tím méně bude vypočítaných izočár ve výstupním vektorovém obrázku.

8 Závěr

Cílem této práce bylo v prvé řadě nastudování základních principů a postupů při tvorbě aplikací za využití technologie CUDA tak, abych byl schopen v další fázi po seznámení se s daným problémem generování izočár pomocí algoritmu Marching squares naprogramovat aplikaci, která by s využitím výkonu grafické karty dokázala paralelně zpracovat vstupní data z vloženého obrázku s výškovou mapou.

Výsledná aplikace splňuje definované požadavky. Dokáže tedy načíst vstupní obrázek ve formátu BMP, ohodnotit jeho pixely a tyto hodnoty použít pro generování izočár. Nakonec jsou izočáry vyexportovány do SVG formátu vektorové grafiky.

Výhodou této aplikace je její velmi rychlé generování izočár prováděné grafickou kartou. Nevýhodou je však část kódu zpracovávaná procesorem, která výrazně zpomaluje celou aplikaci. Jedná se především o export vygenerovaných izočár do vektorové grafiky. Pokud se totiž v obrázku vyskytuje mnoho různorodých oblastí (časté přechody úrovní barev) nebo jiným způsobem zvyšuje náročnost na počet vygenerovaných izočár (například velkými rozměry), vygenerovaných izočár je spousta a trvá neúměrně dlouhou dobu, než se tyto čáry postupně uloží do výstupního souboru představujícího obrázek vektorové grafiky. Navíc takovýto obrázek má velikost v řádech desítek megabytů a následně se i pomalu otevírá v příslušném programu pro prohlížení či editaci vektorové grafiky.

Návrhů na vylepšení aplikace je hned několik. V prvé řadě by se dala rozšířit podpora vstupních obrázků a to nejen z hlediska formátů, ale i barevné hloubky (využití nejen stupňů šedi, ale všech RGB složek). Dále by se dala zpracovat podpora více formátů pro export izočár do vektorové grafiky nebo upravit stávající export tak, aby byl méně náročný jak na dobu zpracování, tak na velikost výstupního souboru.

Využití programu bych viděl ve zapracování algoritmu pro výpočet izočár a jejich uložení v paměti do pokročilejší aplikace, která těchto výpočtů využívá. Mohlo by se jednat například o aplikaci, která by vypočítávala určitou deformaci terénu na základě dodané vstupní výškové mapy.

Jak je i z výsledků výkonnostních testů patrné, zpracováním výpočetně náročných aplikací pomocí grafických karet s podporou technologie CUDA dochází k razantnímu zrychlení v řádech desetinásobků oproti rychlosti zpracování s využitím současných procesorů. Proto si myslím, že technologie CUDA najde v brzké budoucnosti hojné uplatnění u aplikací náročných na výpočetní výkon. Mezi ně se řadí hlavně aplikace používané ve vědě a výzkumu, kdy již nebude zapotřebí velké množství speciálně navržených sálových počítačů pro tyto účely, ale bude možné takovéto výpočty provádět na obyčejných osobních počítačích. Nebo vzniknou nové sálové počítače, které ovšem nebudou sestaveny z několika procesorů, ale budou sestaveny z několika grafických karet a hranice maximálního dosaženého výkonu se opět posune o velký kus dále.

Uplatnění by tato technologie našla samozřejmě i v programech pro širokou veřejnost, jako jsou například editační nástroje pro hudbu, grafiku (2D i 3D) či video. Ovšem jak jsme se již mohli několikrát v historii přesvědčit, výrobci softwaru neradi mění zaběhané zvyklosti. Více jádrové procesory jsou na trhu již několik let, ale softwaru, který by byl schopen takový výpočetní výkon využít, za tu dobu příliš nepřibylo. Navíc aplikace

vyvíjené s využitím technologie CUDA lze provozovat pouze na grafických kartách s čipem společnosti nVidia, tudíž takový software nebude dostupný pro celou širokou veřejnost, ale jen pro její část. A proto si myslím, že prozatím bude mít technologie CUDA své hlavní pole působnosti pouze v odvětví vědy a výzkumu.

9 Reference

[1] HUGHES, Cameron - HUGHERS, Tracey. *Parallel and distributed programming using C++*, 2004. ISBN 0-13-101376-9

[2] HUGHES, Cameron - HUGHERS, Tracey. *Professional Multicore Programming: Design and Implementation for C++ Developers*, Indiana: Wiley Publishing Inc., 2008. ISBN 978-0-470-28962-4.

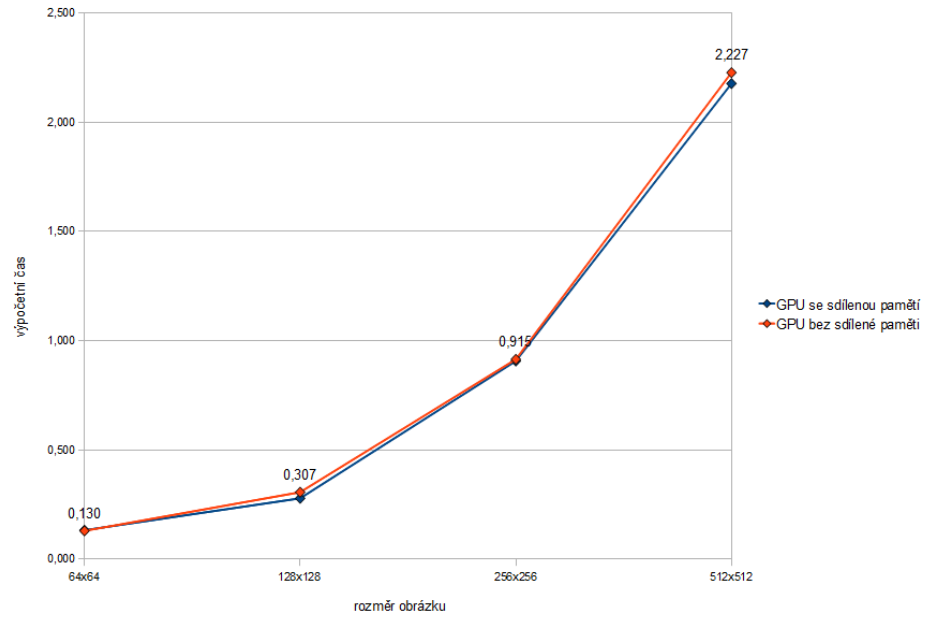
[3] HERLIHY, Maurice - SHAVIT, Nir. *The Art of Multiprocessor Programming*, 2008. ISBN 978-0-12-370591-4.

[4] ČERVENÝ, Jiří - ŠIMEK, Petr. *Marching Cubes* [online], 1998 [cit. 2010-04.29]. Dostupné z: <http://iris.uhk.cz/grafika/mcb/mc0.htm>

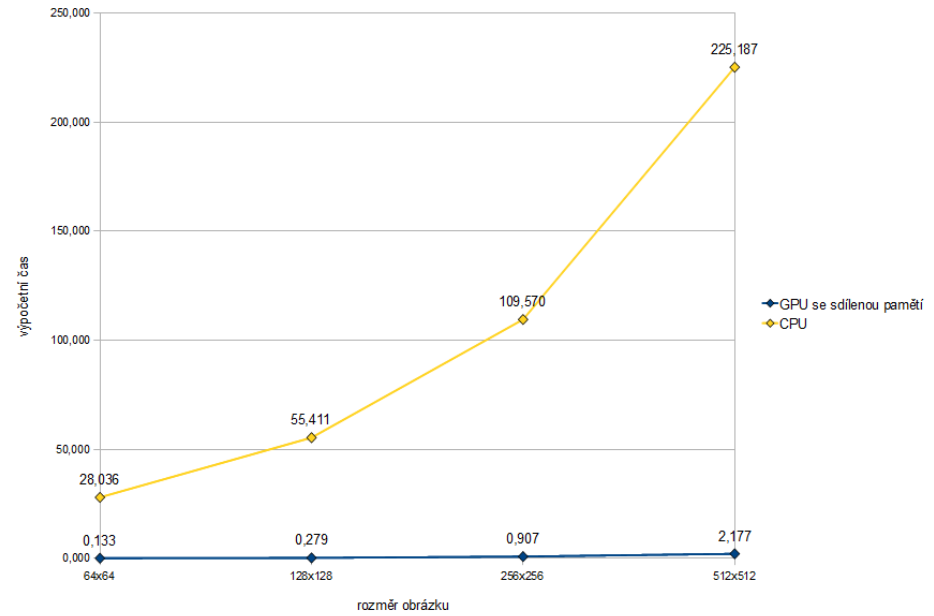
[5] FARBER, Rob. *CUDA, Supercomputing for the Masses: Part 4* [online], June 03, 2008 [cit. 2010-04-29]. Dostupné z: <http://www.drdobbs.com/architecture-and-design/208401741>

[6] *NVIDIA CUDA Programming Guide* [online], NVIDIA, 2010.
Dostupné z: <http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf>

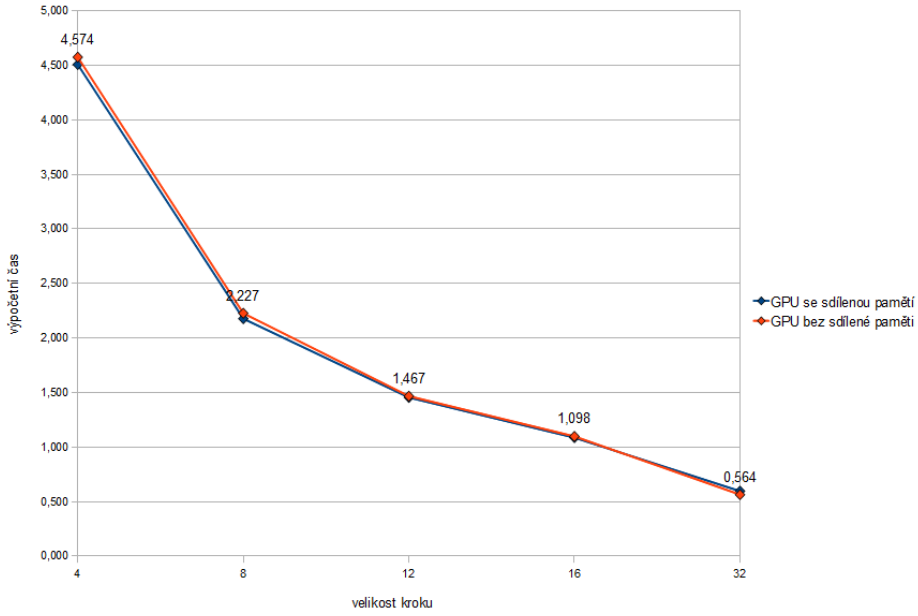
A Grafy výsledků testů



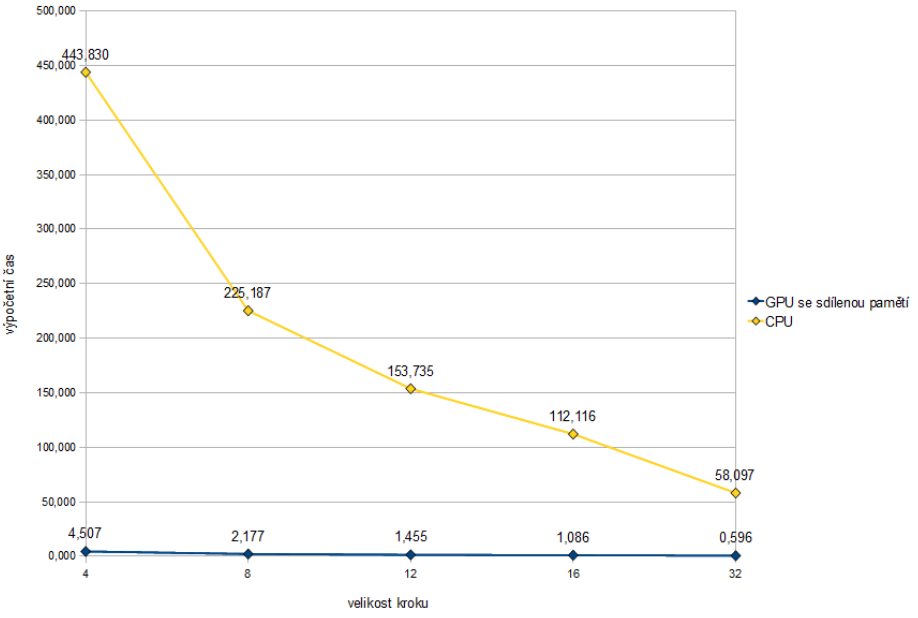
Obrázek 11: Srovnání pamětí GPU při různých rozměrech obrázku.



Obrázek 12: Srovnání GPU a CPU při různých rozměrech obrázku.



Obrázek 13: Srovnání pamětí GPU při různých velikostech kroku.



Obrázek 14: Srovnání GPU a CPU při různých velikostech kroku.